Code Cleaning Utility

An Application Guide

Table of Contents

Introduction

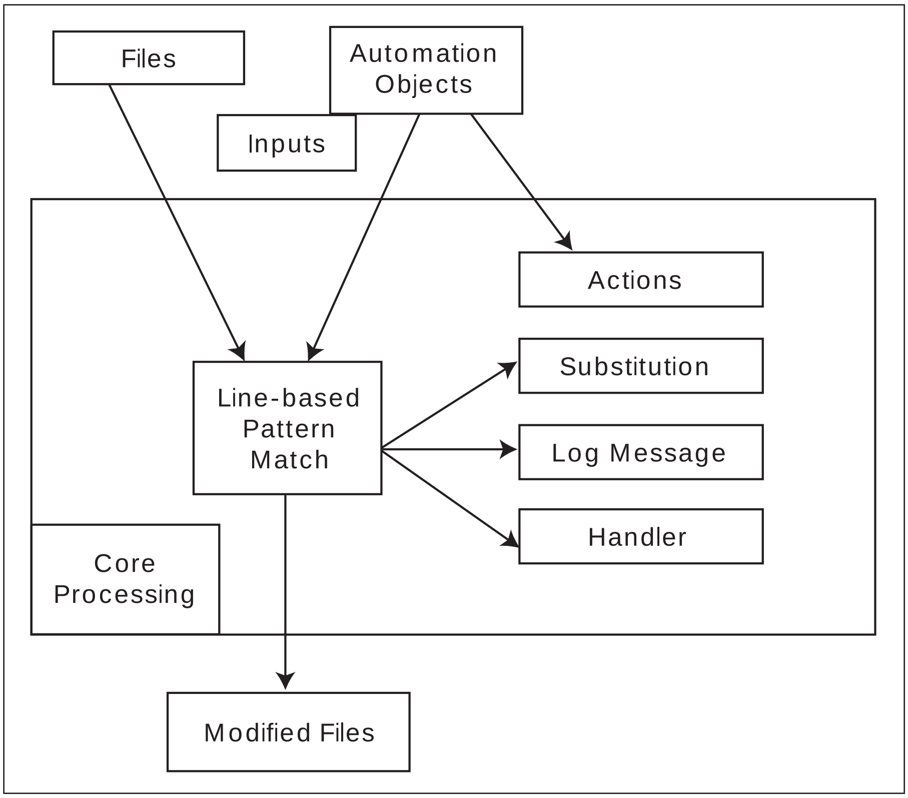
The IA-64 instruction set architecture introduces larger pointer and integral values which enable addressing large address spaces. This new enhancement means new programming models for the application programmer. The ILP32 programming model, predominant in the computer industry and on all IA-32 operating systems, now makes way for P64 and LP64 programming models. Preparations to update code from the ILP32 programming models to take advantage of the large address space and longer pointers have been underway at many ISVs for some time now. This preparation is sometimes referred to as code cleaning.

Some code cleaning can be automated, as in the case of a change in API. These API changes may be uniform across the entire application or may be conditional, depending on the context of the code. Other modifications are more difficult to automate, such as detecting and correcting 64-bit and 32-bit integral datatype mismatches. A pattern matching tool is capable of addressing the uniform and conditional changes in API, while a compiler parser tool is capable of addressing datatype mismatches.

This application note introduces the Code Cleaning Utility, a utility that uses pattern matching to detect common code problems and then take action. The utility does not include a compiler parser and is unable to detect type mismatches unless they can be recognized using a pattern. The utility is designed for flexibility and customization, so that you can set up automation solutions specific to your application APIs, and based on issues unique to your porting experience. The application note describes this utility's usage and how it can be customized.

For convenience and for purposes of example, this utility already contains some standard conversions for the Win64* programming model. These standard conversions apply to all current Win32* applications. The inclusion of these conversions by no means limits this tool to Windows* applications. As a pattern matching utility, there is no inherent bent toward any platform. Customize this tool for your application, whether for Windows or for Unix*, adding and removing conversions as appropriate.

This application note is organized in the following manner. One section describes the code cleaning utility and how it is invoked. Another section describes how it can be extended and customized. A resources section provides pointers to additional resources to be used for customization trouble shooting. The application note wraps up with conclusions.



Code Cleaning Utility

The Code Cleaning Utility solves part of the problem related to automating the port of an application to Intel® Itanium™ processor-based platform. It takes source files and Automation Objects as input. Automation Objects contain information specifying the pattern to be searched for and the automation action to be taken once the pattern is found. Within the core processing, the utility opens each file and performs line-based pattern matching using information from the Automation Objects. When a match is found, the utility invokes the specified action. Actions include substitution of a new string for the target pattern; logging the filename, the line number, and an informative message; and invoking a handler to take action on the file. Each of the actions is configurable by the user. The invocation of a handler is particularly powerful, since you can write the handler yourself.

Statistics are gathered and change logs are recorded as the application code is processed, so that you can see the actual changes in the end. These statistics give you additional insight by summarizing the location of issues and the number of occurrences. With the handlers that you write, you are given the opportunity to gather and report your own statistics. The utility includes one more feature that allows you to make a pass over your code (gathering statistics and generating reports) without making any actual modifications to the code. You can use this feature to assess the state of your code, without committing to the changes.

As previously mentioned, the utility contains some standard conversions for the Win64 programming model. These are based on the article “Getting Ready for 64-bit Windows” that can be found in the current Microsoft Platform SDK documentation. In particular, the API changes noted in the section “Win32 API Changes” are addressed. For those porting Unix applications, consider removing these specific conversions in case they conflict with your own application interfaces.

Running the Code Cleaning Utility

Here is a description of how to run the utility from the command line.

```
prompt> ia64tool.exe <filename> <"Test" or "Modify">
```

If Python is installed (http://www.python.org/download):

```
prompt> python ia64tool.py <filename> <"Test" or "Modify">
```

where <filename> is the name of a file that contains a list of the source files you want to have examined. The default is not to make modifications. If "Modify" is specified, then the utility will make changes. See the Output section for details on the files that are generated by the utility.

Customizing the Code Cleaning Utility

Customization of the Code Cleaning Utility is contained in the file "ia64tool_usrcfg.py". This file is divided into three sections: substitution, log, and handler. In the substitution section, you specify the pattern to search for and the string that substitutes for the pattern. In the log section, you specify the pattern to search for and the informative message that should be printed out when the pattern is found. In the handler section, you specify the pattern to search for and you provide the handler that the utility will invoke when the pattern is found. Examples can be found in the file mentioned above. Each action is discussed in the sections below.

Substitution

In the substitution area, the tool makes use of "subspkg" for substitutions. "subspkg" contains two lists, one for basic substitutions and one for user specific substitutions. You use the latter list "user_specific_substitution" for your application specific customizations. The expected format is as follows:

```
user_specific_substitution=[
    ( pattern1, case sensitivity, replace string1 ),
    ( pattern2, case sensitivity, replace string2 ),
    ...
    ]
```

Case sensitivity pertains to the pattern. Options include CS for "case sensitive" searching and CI for "ignore case" searching.

```
For example:
user_specific_substitution=[
    ( r"\bGCL_WNDPROC\b", CS, "GCLP_WNDPROC" ),
    ( r"\bGetWindowLong\b", CS, "GetWindowLongPtr" )
    ]
```

This is a list containing directions for two case sensitive substitutions. The patterns are regular expressions. "\b" at the beginning and end of the pattern specifies word boundaries (this will only match whole words). The "r" tells Python to treat this as a raw string, allowing the use of "\" without having to escape it "\\". Take care with backslash escapes (e.g., "\n") in replacement strings, since they will be interpreted according to their escaped meaning. See the Resources section below for regular expression resources to use for trouble shooting.

Log Message

In the log section, the tool makes use of “flagpkg” for logging informative messages. “flagpkg” contains two lists, one for standard log messages and one for user specific log messages. You use the latter list “user_flag_items” for your application specific customizations. The log message section looks similar to the substitution section, except that the 3rd argument in the pattern directions is an informative string that is logged when the pattern is found, along with the location of the occurrence. In the “ia64tool_usrcfg.py” file, the log items are referred to as items to be “flagged”. The expected format for the “user_flag_items” list is as follows:

```
user_flag_items=[
    ( pattern1, case sensitivity, informative string1 ),
    ( pattern2, case sensitivity, informative string2 ),
    ...
    ]
```

The case pattern and case sensitivity fields are the same as for substitution. The informative string is simply a string.

```
For example:
user_flag_items=[
    ( "\([\w_]+\)\s*[lw]param", CI,
    "Flag: lparam or wparam casting occurring" ),
    ( "if.*WIN32", CS, "Flag: Examine this WIN32 code guard" )
    ]
```

Handler

In the handler section, the utility makes use of “handlerpkg” for invoking handlers. “handlerpkg” contains two lists, one for standard handlers (currently none) and one for user specific handlers. You use the latter list “user_handlers” for your application specific customizations. The expected format for “user_handlers” is:

```
user_handlers=[
    ( pattern1, case sensitivity, instance of a handler class ),
    ( pattern2, case sensitivity, instance of a handler class ),
    ...
]
```

Pattern and case sensitivity are the same as substitution and logging messages.

The handler class is the only section that requires some knowledge of programming in Python. See the Resources section for a list of references available to help you learn what you need to know.

Assuming that you have some knowledge of C++ object-oriented programming, you will understand the following description of how handlers work with this utility. This utility includes an abstract class “handler_abstract_class” that you use to build your handler. There are three member functions in the class: the constructor “__init__”, the handler “handler”, and “finalize” for post processing and reporting. Here is the abstract class.

```
Class handler_abstract_class:

    def __init__(self):
        # Dictionary to hold statistics
        self._stats={"Line count":0,
                     "File count":0}

    def handler(self, regex_match, fname, lines):
        """
        This routine is called when there is a match
        by regular expression. fname and regex_match are
        passed in for book keeping purposes. lines must
        be returned.
        """
        print "In handler_abstract_class::handler (%d)%s"\
        %(len(lines),fname)

        # Gather statistics
        self._stats["File count"]=\
            self._stats["File count"]+1
        self._stats["Line count"]=\
            self._stats["Line count"]+len(lines)

        for i in range( len(lines) ):
            # Search line by line for pattern (use re module)
            # Change line if necessary
            # This example only prints a message
            # You may also want to keep stats in here
            l=lines[i]
            s = "%s(%d): Cast of \"this\" " % (fname, i+1)
            s + s+"pointer is occurring\n\t%s"%l
            if re.search(r"\(([\w\s]+?)\)\s*this\b",l):
                    print s

        # Return modified lines
        return lines

    def finalize(self):
        """
        This routine is called at the end of the program
        and could be used for reporting info and statistics
        that you collected in handler.
        """
        print "In handler_abstract_class::finalize"

        s="Statistics:\n"
        s=s+"File count: %d\nLines count: %d\n"%\
        (self._stats["File count"], self._stats["Lines count"])

        # Open file, writes to it, and closes the file
        open("Default_log.txt","w").write(s)
```

Place initialization code in __init__(). Use finalize() to print info and statistics that you have collected along the way. finalize() is called in the report phase of the tool afterall code has been processed.

handler() is called for each file that has a line matching the corresponding pattern. It is only called once for a file, even though there may be multiple lines that match the pattern. Therefore, handler() should take care of all issues related to the pattern in that one call. handler() only operates on “lines”. It would be incorrect for it to open the file and write to it directly, since there may be other substitutions or handlers that are also making changes. The “lines” parameter contains the cumulative changes made by all substitutions and other handlers. handler() makes its changes and returns the modified lines as output. The file name and the pattern are passed to handler() for bookkeeping purposes, but are not required.

Output

The output from the utility includes output to the console, standard reports written to files, and reports that you may generate in the handlers that you write. Almost all results are in tab-delimited form to allow them to be imported into a spreadsheet for analysis or reformatting.

Output to the console begins with a progress list consisting of each filename examined along with its corresponding number of lines in the file. If you are running the utility in test mode, files that will be changed are identified with the message “Testing. Will be overwritten when not testing.” On the other hand, running in “Modify” mode generates the message “Overwriting”. Here are some examples of console output. No comment indicates no change to the code.

```
(1374)./myapp/myappmain.Cpp : Overwriting.
(131)./myapp/sync.Cpp
(127)./myapp/runtime.Cpp
(145)./myapp/timer.Cpp
(410)./myapp/realtime.Cpp
(479)./myapp/collector.Cpp
```

A summarization of substitutions and flagged items follows the progress listing.
Here is an example:

```
Files Examined: <count>
Lines Examined: <count>
<date>

Summary Substitutions:
    count1 pattern1
    count2 pattern2
    ...
Detail written to file: substitution_detail.rpt

Summary Flag Items:
    count1 Description1
    count2 Description2
    ...
Detail written to file: flagged_items_detail.rpt

Summary Handlers:
    count1 Handler1
    count2 Handler2
    ...
```

In the case of the Summary Handlers section, the counts represent the number of files upon which the utility invoked the handler. It does not necessarily indicate the number of files that were changed by the handler, since the handler may have examined the file and decided to make no changes at all.

The table below shows additional reports that are written to a file along with their output formats.

File Name	Generated From	Output
substitution_detail.rpt	This report is generated based on substitutions performed by the utility.	<date> Summary Substitutions: <count1> <pattern1> <count2> <pattern2> ... Detail Substitutions: <file1> <line1> <pattern> <file2> <line2> <pattern> ...
flagged_items_detail.rpt	This report is generated based on items that were flagged by the utility.	<date> Summary Flag Items: <count1> <descriptive message> <count2> <descriptive message> ... Detail Flag Items: <file1> <line1> <descriptive message> <file2> <line2> <descriptive message> ...
Remaining report files.	These are generated by the default handlers or by handlers that you may have written. File names are defined by the handler as well as the file content. Printing the name of the file to the console along with a description would be useful run-time information.	Output is handler specific, but should include a summary section followed by a detail section.

Resources

Python

The Code Cleaning Utility is written in the Python programming language (http://www.python.org), an Open Source programming language that is Copyrighted, but freely distributable even for commercial use. Python is required to run this utility. Since Intel does not distribute Python binaries, here are the instructions for downloading and installing Python yourself.

- Download py152.exe from the Python web site http://www.python.org/download/download_windows.html

  This executable installs the core Python interpreter along with supporting modules.

- Download win32all-128.exe from Mark Hammond's Python web site http://starship.python.net/crew/mhammond/win32/Downloads.html

  This installs Python extensions specifically targeted at Microsoft Windows* platforms.

- In the command window that you use to run this utility, add the directory containing python.exe to your path.

This language is available on Windows and Unix platforms. If you are interested in writing your own handler, you may need a quick introduction to Python. To assist you, take a look at these resources at http://www.python.org/doc/Intros.html, (recommended in the order listed).

- Instant Python
- Getting Started With Python
- Python Tutorial
- Python Online Documentation (http://www.python.org/doc)

Regular Expressions

If you are unfamiliar with regular expressions or if you are familiar with Perl's regular expression syntax and would like to understand Python regular expressions, then look at the Python regular expression HOWTO guide at http://www.python.org/doc/howto. Also, take a look at the Python documentation on the "re" module at http://www.python.org/doc.

To be very clear about how the regular expression substitution works, the following Python code is equivalent to what actually will occur for each line substitution. Recall that substitution is just one of three possible actions available to you. The example below is included because regular expressions are very powerful and you can do much more than is described in this note. In fact, you can do anything that these two lines and the Python "re" module will allow (i.e. the replace_string could be a string, a regular expression, or a even a function). See the resources above for all the details.

```
# Compile the pattern in advance
compiled_match_object = re.compile( pattern [, re.IGNORECASE] )
# Replace "pattern" with "replace_string" in "line".
# Return modified "line".
line = compiled_match_object.sub ( replace_string, line )
```

Conclusion

In summary, the Code Cleaning Utility provides a flexible means of automating portions of the code cleaning process of moving to Intel® Itanium™ processor-based platform. The utility uses pattern matching and the option of several different actions to support the automation process. You can customize the actions of substitution, logging a message, and applying a handler. The handler, in particular, puts maximum flexibility at your fingertips to tackle more complex automation tasks.

Information in this document is provided in connection with Intel® products. No license, express or implied, by estoppel or otherwise, to any intellectual property rights is granted by this document. Except as provided in Intel's Terms and Conditions of Sale for such products, Intel assumes no liability whatsoever, and Intel disclaims any express or implied warranty, relating to sale and/or use of Intel products including liability or warranties relating to fitness for a particular purpose, merchantability, or infringement of any patent, copyright or other intellectual property right. Intel products are not intended for use in medical, life saving, or life sustaining applications. Intel may make changes to specifications and product descriptions at any time, without notice.

*Third-party brands and names are the property of their respective owners.
Copyright © Intel Corporation, 2000. All rights reserved.